Implementing Circular Buffers With Bit-Reversed Addressing

APPLICATION REPORT: SPRA292

Henry D. Hendrix
Senior Member Technical Staff

Digital Signal Processing Solutions
November 1997

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain application using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

TRADEMARKS

TI is a trademark of Texas Instruments Incorporated.

Other brands and names are the property of their respective owners.

CONTACT INFORMATION

US TMS320 HOTLINE	(281) 274-2320
US TMS320 FAX	(281) 274-2324
US TMS320 BBS	(281) 274-2323
US TMS320 email	dsph@ti.com

Contents

Figures

Tables

Implementing Circular Buffers With Bit-Reversed Addressing

Abstract

Delay lines are used in many DSP applications. Delay lines provide a buffer of the last N samples of data for a filter or other algorithm. While delay lines can be implemented as linear or circular buffers, buffers of the circular variety are often preferred due to their relative simplicity.

Texas Instruments' (TI) TMS320 family of DSPs can manipulate circular buffer pointers without penalty to code size or execution time through their bit-reversed addressing capabilities. Through consistent pointer manipulations, these buffers can be used anywhere traditional circular buffers are used.

This application note describes:

- How linear and circular buffers handle data movement in a delay line.
- The properties and limitations of circular buffers.
- How to implement circular buffers with bit-reversed addressing.
- Three methods, with step-by-step examples, for handling pointer manipulations.

Product Support

World Wide Web

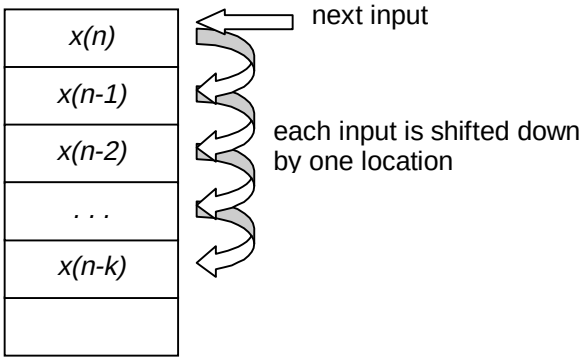
Our World Wide Web site at www.ti.com contains the most up to date product information, revisions, and additions. Users registering with TI&ME can build custom information pages and receive new product updates automatically via email.

Linear Versus Circular Buffers

Linear Buffers

Linear buffers require that the delay be implemented by manually moving data down the delay line. The new data is written to the recently vacated spot at the top of the buffer.

Figure 1: Linear Buffer Requires Manual Data Movement

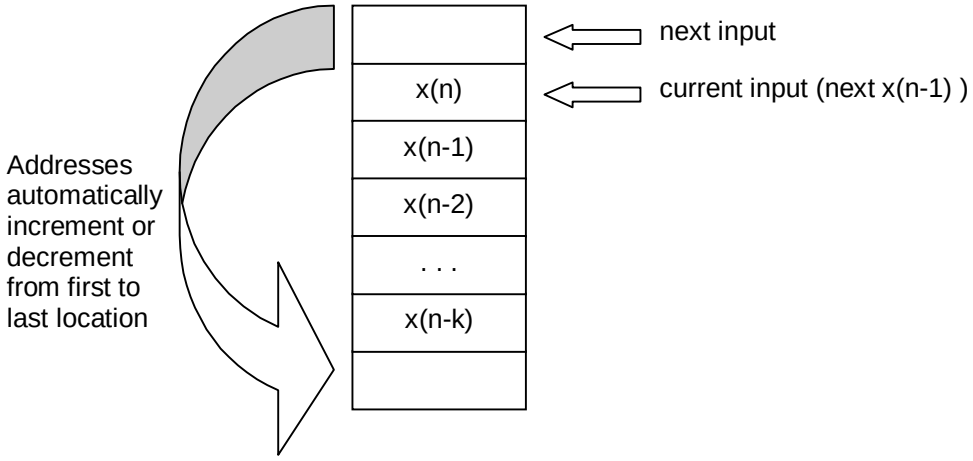


Although TMS320 DSPs can perform this data movement in parallel with arithmetic processing, use of this feature is limited to the on-chip memory. Manual data movement in external memory is discouraged since it involves external data writes, which typically require at least 3 cycles. The parallel data moves are also limited in movement by only one data location and in only one direction. Some applications, such as a decimating filter, require the sample data to be accessed by an offset greater than one.

Circular Buffers

Circular buffers implement the delay line by moving a pointer through the data, rather than moving the data itself. New data is written one position above the previous sample. This requires that the pointer must be able to jump from the last location to the first, or vice-versa. In this manner, the buffer appears to be continuous, with the newest data overwriting the oldest data.

Figure 2: Circular Buffer Moves Pointer Through Data



Since a great deal of software overhead is involved in checking the pointer values after every update, many DSPs provide circular buffering hardware that performs this function automatically. This hardware will wrap the pointer around once it goes past the end or start of the buffer.

Although some of the TMS320 DSP family members (namely the C1x, C2x, and C2xx) do not provide this hardware, they can still provide zero-overhead circular buffering by using their bit-reversed addressing hardware, normally used in FFT algorithms. This same method can also be used in DSPs with circular buffering hardware to provide additional circular buffers.

Theory

The only requirement for implementing a circular buffer is a method to keep a pointer's value within the range of the buffer and wrap around from the highest to lowest address (or vice-versa).

Using normal pointer arithmetic, carries are propagated to the left, causing the values to grow until the highest 16-bit number is reached.

With bit-reversed addressing, the pointer arithmetic is done such that the carry bits propagate to the right. Carries from the rightmost bit are ignored, thus confining the pointer to a range determined solely by the value used to increment or decrement the pointer. As long as the pointers are updated in bit-reversed manner, only the specified lsb's are ever modified.

The following example will illustrate this concept. Given a buffer size of 8, an index register set to 4, and an initial pointer set to 0x100, the sequence of accesses to the buffer is shown in Table 1.

Table 1: Buffer Access Sequence

Accesses	Address (hex)	Address calculation of 3 lsb's	Comment
start	0x100		
*BR0+	0x104	000 + 100 = 100	
*BR0+	0x102	100 + 100 = 010	carry propagated to the right!
*BR0+	0x106	010 + 100 = 110	
*BR0+	0x101	110 + 100 = 001	
*BR0+	0x105	001 + 100 = 101	
*BR0+	0x103	101 + 100 = 011	
*BR0+	0x107	011 + 100 = 111	
*BR0+	0x100	111 + 100 = 000	carry falls off the right side!

Limitations

From this example we can see that a few restrictions are required to use this method.

1) The size of the buffer must be a power of two (2^n). The filter length can be any size, however (see examples to follow).

2) The buffer must be aligned so that the starting address of the buffer has *n* lsb's equal to zero.

3) All pointer updates must use the bit-reversed update mode (*BR0+/-). This requires that the desired increment/decrement value be stored in the index register (AR0) in bit-reversed form.

NOTE:

All 2^n data values in the buffer are accessed but not in linear order. This means that all accesses to the data in the buffer *must* comprehend the ordering of the data and never modify the circular buffer pointers using normal updates.

Methodology

Given the properties and limitations above, the following procedure should be used to implement a circular buffer:

1) Determine the required buffer size. This should be a number that is the smallest power of two that will hold all the data. For example, a buffer of 48 would require a buffer of size 64 (2^n, where n=6).

2) Align the buffer such that the *n* lsb's are zero. This can be done manually or by using the "align" directive in the linker command file.

3) Initialize a pointer somewhere within the buffer. While typically you would set the pointer to the start of the buffer, it will work as long as it is anywhere within the buffer.

4) Set the index register (AR0 for the C2xx) to the desired offset value, *in bit-reversed form*. Typically an offset of 1 is used, which is half the buffer size in bit-reversed form. For example, a size 64 buffer would set AR0 = 32, or 10 0000 binary.

5) Step through the buffer using bit-reversed mode (*BR0+/-) to update the pointer.

The following example shows code implementing a simple filter on a C203 in which the filter length is the same size as the circular buffer. Note that the data pointer ends up at the starting point of the buffer and must be manually moved back one position in order to implement the delay function.

Example 1: FIR Filter of Length 16

```
FRAME  .set  50             ; number of samples
       .bss  cbuf,16        ; circular buffer of size 16
       .bss  temp,1         ; temporary memory location

COEF   .sect “coef”         ; coefficients in program memory
       .word 3129h          ; (not all shown)
       .word 7422h

       .text
setup  lar   ar0,#8         ; ar0 = 1000b (bit-reversed 1)
       lar   ar1,#cbuf      ; ar1 points into “cbuf”
       lar   ar2,#FRAME-1   ; ar2 is loop counter
       mar   *,ar1          ; set auxiliary register pointer to ar1
       ldp   #temp          ; set data page pointer to “temp” page

filter in    *,2            ; get input from I/O port 2
       mpy   #0             ; clear P reg
       lacl  #0             ; clear ACC
       rpt   #15            ; do 16 taps
        mac  COEF,*BR0+     ; ACC += x(n-k)*h(k)
       apac                 ; do last accumulate
       sach  temp,1         ; save result (Q15)
       out   temp,5         ; output to I/O port 5
       mar   *BR0-,ar2      ; move pointer back 1 & change ARP to loop counter
       banz  filter,*-,ar1  ; loop until done
```

Pointer Manipulations

There are often additional pointer manipulations to be made after the filtering or other function is complete. For instance, when using a filter length that is less than the buffer size, resetting the pointer to the location of the next input sample requires movement by a different value than used in the filter. Another example is access of an arbitrary sample in the delay line. Three different methods for manipulating bit-reversed pointers are shown below.

Method 1: Computing Offset Values

Since the number of pointer increments through a filter is known, a number can be calculated to return the pointer to the location for the next input. Maintaining all accesses in bit-reversed form requires that this offset be in bit-reversed form as well. The procedure is as follows:

1) Compute the offset value.

   For example, after processing a 40-tap FIR filter, the pointer must be rewound by 41 to implement the delay. An offset value of 41 (decimal) = 29 (hex) = 10 1001(binary).

2) Bit-reverse this to get 1001 01 (binary) = 25 (hex) = 37 (decimal).

3) Load the index register (AR0) with the bit-reversed offset value.

4) Subtract this value from the pointer: “mar *BR0- “.

5) Reset the index register (AR0) to its original value of bit-reversed one: 10 0000 (binary) = 20 (hex).

It is simplest to compute the offset as a positive number; then either add or subtract it from the pointer. While negative offsets can be used, they are limited to half the buffer size.

Example 2 shows the same filter implementation as example 1, but with a tap length of 40, which uses only part of the circular buffer. Note that after the filter, the pointer is at the 41st sample; thus, it must be moved back by 41 words to get to the correct spot for the next sample. This method requires 5 cycles on the C203 for the pointer update.

Example 2: FIR Filter of Length 40, Computed Pointer Update

```
FRAME  .set  50             ; number of samples
       .bss  cbuf,64        ; circular buffer of size 64
       .bss  temp,1         ; temporary memory location

COEF   .sect “coef”         ; coefficients in program memory
       .word 3129h          ; (not all shown)
       .word 7422h

       .text
setup  lar   ar0,#20h       ; ar0 = 10 0000b (bit-reversed 1)
       lar   ar1,#cbuf      ; ar1 points into “cbuf”
       lar   ar2,#FRAME-1   ; ar2 is loop counter
       mar   *,ar1          ; set auxiliary register pointer to ar1
       ldp   #temp          ; set data page pointer to “temp” page

filter in    *,2            ; get input from I/O port 2
       mpy   #0             ; clear P reg
       lacl  #0             ; clear ACC
       rpt   #39            ; do 40 taps
        mac  COEF,*BR0+     ; ACC += x(n-k)*h(k)
       apac                 ; do last accumulate
       sach  temp,1         ; save result (Q15)
       out   temp,5         ; output to I/O port 5
       lar   ar0,#25h       ; ar0 = 10 0101b (bit-reversed 41)
       mar   *BR0-,ar2      ; move pointer back 42 & change ARP to loop ctr
       lar   ar0,#20h       ; ar0 = 10 0000b (bit-reversed 1)
       banz  filter,*-,ar1  ; loop until done
```

Method 2: Saving the Original Value

A slightly quicker method for restoring a pointer position is to save the starting address in a temporary variable; then restore it at the end. This procedure is:

1) Save the initial pointer value (i.e., the address of the current sample).

2) Perform the filter calculations.

3) Restore the pointer to the initial value.

4) Decrement the pointer by one (bit-reversed) to point to the next sample.

Example 3 shows an example of this method, using the same filter implementation as examples 1 and 2. This method requires 4 cycles on the C203 for the pointer update. Sometimes the last decrement can be combined with another function to further reduce the cycle count.

Example 3: FIR Filter of Length 40, Saved Pointer Update

```
FRAME  .set  50             ; number of samples
       .bss  cbuf,64        ; circular buffer of size 64
       .bss  temp,1         ; temporary memory location
       .bss  temp2,1        ; temporary memory location

COEF   .sect "coef"         ; coefficients in program memory
       .word 3129h          ; (not all shown)
       .word 7422h

       .text
setup  lar   ar0,#20h       ; ar0 = 10 0000b (bit-reversed 1)
       lar   ar1,#cbuf      ; ar1 points into "cbuf"
       lar   ar2,#FRAME-1   ; ar2 is loop counter
       mar   *,ar1          ; set auxiliary register pointer to ar1
       ldp   #temp          ; set data page pointer to "temp" page

filter sar   ar1,temp2      ; save current data pointer value
       in    *,2            ; get input from I/O port 2
       mpy   #0             ; clear P reg
       lacl  #0             ; clear ACC
       rpt   #39            ; do 40 taps
        mac  COEF,*BR0+     ; ACC += x(n-k)*h(k)
       apac                 ; do last accumulate
       sach  temp,1         ; save result (Q15)
       out   temp,5         ; output to I/O port 5
       lar   ar1,temp2      ; restore data pointer
       mar   *BR0-,ar2      ; move pointer back 1 & change ARP to loop ctr
       banz  filter,*-,ar1  ; loop until done
```

Method 3: Using Different Offset Values

Another possible manipulation requires using different offset values, such as in a decimating filter. For instance, to use every other data point in a filter, the index register would be set to a value of 2 in bit-reversed form, instead of 1. In examples 2 and 3, this would require setting AR0 to 01 0000 (10h) instead of 10 0000 (20h). No other changes are required.

Conclusion

Even when lacking dedicated circular buffering hardware, all members of the TMS320 family of DSPs can implement circular buffers through their bit-reversed addressing capabilities. This capability requires about the same overhead as dedicated circular buffers, both in code size and in execution time. It has been shown that use of these types of circular buffers have minimal limitations. Through consistent pointer manipulations, these buffers can be used wherever traditional circular buffers are used.

References

Randy Restle, “Circular Buffer in Second Generation DSPs”, *TMS320 Designer’s Notebook Page #7*, Texas Instruments, 1996.

TMS320C2xx User’s Guide, Digital Signal Processing Solutions, Texas Instruments, 1995.